CMP3018M Graphics

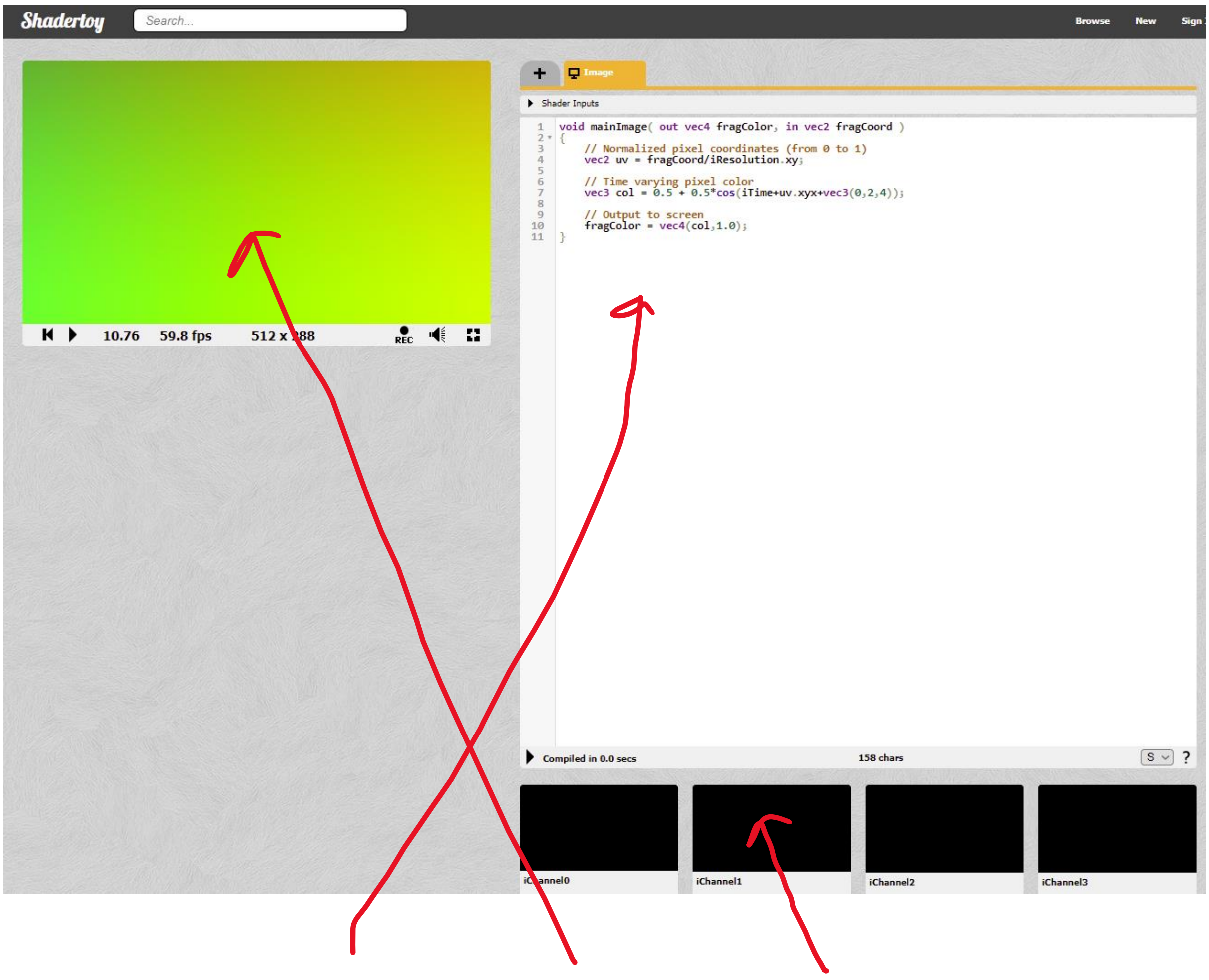
Week 1 Tasks

We will start looking at shaders with ‘Shadertoy’ (**https://www.shadertoy.com/new**).

Shadertoy is an online editor and run environment for fragment shaders. It enables you to experiment with shader code, but not to worry about the application it is run from (Unreal, Unity, OpenGL code, etc).

Task 1:

In order to get familiar with the Shadertoy interface, we will use the default fragment shader which is shown. You will see something like this:



```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

There is the shader code editor pane, the shader output, and ‘iChannel0..3’ which can be used to bring textures into your code.

At the top of the editor pane, there is a drop down called ‘Shader Inputs’. These are all the ‘uniform variables’ which are made available to the shader code by the main program.

Try compiling the shader by clicking the small arrow, then running the compiled shader in the shader output pane.

Select ‘Browse’ to have a look at the impressive shader effects that can be done here.

Task 2:

Change line 7 to:

```
vec3 col = vec3(0.0, 1.0, 0.0);
```

You should see a completely green canvas. This line is setting the pixel colour for every pixel in the canvas in parallel – i.e. at the same time!

Task 3:

Lets try and change the colours of the pixels based on their position on the canvas.

Lets try to make the top half of the canvas blue…

HINT: `uv.y` is the y coordinate of the position of the pixels on the canvas.

Lets go back to the default shadertoy code – refresh the page, and the code should return to what it was.

Task 4:

```
// Time varying pixel colour

vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

fragColor = vec4(col, 1.0);
```

This line contains ‘iTime’. It is a global time variable (time since the shader started running).

‘`uv.xyz`’ is a vec3 representing the coordinates of the pixels on the screen. uv is a vec2 forced to be a vec3.

What happens to the shader if iTime is removed?

Try changing the values on this line and see how the output changes.

How can we make the colour change faster?

Task 5:

Lets add a texture to the shader output.

Select ‘iChannel0’ and under the ‘Textures’ tab, select a texture.

Comment out the line with iTime, and modify your code to contain:

```
vec4 texColour = texture(iChannel0,uv);//Get the pixel at uv from iChannel0

 // Output to screen

 fragColor = texColour;
```

This code samples the texture from the texture in iChannel0 and then outputs it.

Can you create a greyscale texture version of the one you added to iChannel0?

HINT: the ‘texColour’ rgb values can be accessed directly (texColour.r, texColour.g, etc)

Add in the ‘iTime’ line, then see if you can create a colour changing texture…?

What other cool colour changing effects can you create?

Task 6:

Lets add some movement to our texture. By manipulating the pixel coordinates rather than the pixel colour with the iTime variable, we can see a ripple-like effect:

```
uv.y += cos(uv.x*20.0)*0.05*cos(iTime);
```

If we then add the texture in as before, then you should see the texture appearing to wave or ripple.

You can make the ripples act differently by changing where the varying iTime input is made. If we add it into the first cos term, then as the ripples are ‘offset’ by the varying amount, they appear to be moving:

```
uv.y += cos(uv.x*20.0+iTime)*0.05;
```

This is only working in one dimension so far. Can you add some variation to the x dimension so that the ripples act in both directions – maybe unevenly?

You can use this to produce really cheap and basic liquid movements and surfaces.